Pacman: Tolerating Asymmetric Data Races with Unintrusive Hardware*

Shanxiang Qi, Norimasa Otsuki, Lois Orosa Nogueira, Abdullah Muzahid, and Josep Torrellas
University of Illinois at Urbana-Champaign
http://iacoma.cs.uiuc.edu

Abstract

Data races are a major contributor to parallel software unreliability. A type of race that is both common and typically harmful is the *Asymmetric* data race. It occurs when at least one of the racing threads is inside a critical section. Current proposals that target them are software-based. They slow down execution and require significant compiler, operating system (OS), or application changes.

This paper proposes the first scheme to tolerate asymmetric data races in production runs with negligible execution overhead. The scheme, called *Pacman*, exploits cache coherence hardware to temporarily protect the variables that a thread accesses in a critical section from other threads' requests. Unlike previous schemes, Pacman induces negligible slowdown, needs no support from the compiler or (in the baseline design) from the OS, and requires no application source code changes. In addition, its hardware is relatively unintrusive. We test Pacman with the SPLASH-2, PARSEC, Sphinx3, and Apache codes, and discover two unreported asymmetric data races.

1. Introduction

Data races are arguably the most common type of concurrency bug. They occur when two threads access the same variable without any intervening synchronization and at least one of the accesses is a write. Debugging data races can be notoriously hard and, as a result, there is much research in this area (e.g., [7, 17, 18, 27]). In practice, it is easy to get bogged down uncovering the large majority of the races that are relatively harmless [8, 18] (so-called benign races) at the expense of the harmful ones that cause program crashes, machine hangs, or incorrect program results.

One class of data race that is both common and likely harmful is the *Asymmetric* data race. It occurs when at least one of the racing threads is inside a synchronization-protected critical section [23]. In this case, while a thread (call it *safe*) is accessing shared variables inside a critical section, a second thread (call it *unsafe*) races in, corrupting the state or reading inconsistent state. For example, Figure 1 shows a case where thread T1 is the safe thread. These races are common in bug reports, and can often appear in well-tested codes that interact with third-party or legacy routines [23]. They are likely harmful because the data being corrupted is critical data already protected by synchronization. Interestingly, these races have received little attention [22, 23].

The conventional approach to cope with data races is to detect and remove them through extensive in-house testing. A complementary approach is to *tolerate* the remaining races during production runs. This approach includes techniques that prevent the race from manifesting or that modify the interleaving in a way that minimizes the chances of it (e.g., [21, 28, 30]). This approach is attractive because, even after extensive in-house testing, races have been shown to remain in the code after deployment. Moreover, even for harmful bugs, it takes a long time between the detection of the bug in the field and the release of a fix by the manufacturer [29]. In the meantime, tolerating the race would be beneficial.

```
            T1                          T2
Lock
if (point != NULL){
    point->x = X1;     <-----    point = NULL;
    point->y = X2;
}
Unlock
```

Figure 1. Example of an asymmetric data race.

Asymmetric data races are good candidates for race-tolerance. Indeed, the structure of the race already suggests a way to minimize the potential harm of the race: prevent the unsafe thread from corrupting the state or reading inconsistent state while the safe one is in the critical section. This technique is attractive because, unlike many race-tolerance techniques, it requires no correct-run training. Moreover, for those asymmetric races caused by third party or legacy code interfering with well-tested code, race-tolerance may be the only option, as the unsafe thread code may be unavailable.

There are only two proposals that specifically target asymmetric data races: ToleRace [23] and ISOLATOR [22]. They both use race tolerance and are software-based. When the safe thread enters a critical section, the software makes a copy of the data in the critical section and redirects the safe thread's accesses to the copy. In addition, it may also protect the accesses to the page that contains the original data. Unfortunately, these approaches slow down execution and require significant compiler, operating system (OS), or application changes.

To address these issues, this paper proposes the first scheme to tolerate asymmetric data races in production runs with *negligible* execution overhead. The scheme, called *Pacman*, exploits cache coherence hardware to temporarily protect the variables that a thread accesses in a critical section from other threads' requests. Unlike prior schemes for asymmetric races, Pacman induces negligible slowdown, needs no support from the compiler or (in the baseline design) from the OS, and requires no application source code changes — although small changes are needed in some libraries. Pacman's hardware is largely unintrusive, since it is concentrated in a module in the global network, rather than in the cores. Finally, Pacman embodies a primitive that can be applied to other software development and debugging uses.

We evaluate Pacman for the SPLASH-2, PARSEC, Sphinx3, and Apache codes. We show that it has negligible execution overhead. Moreover, we uncover two unreported asymmetric races.

* This work was supported in part by the National Science Foundation under grant CCF-1012759 and by Intel under the Illinois-Intel Parallelism Center (I2PC). Norimasa Otsuki is with Renesas Electronics Corporation, Japan. Lois Orosa Nogueira is with Universidade de Santiago de Compostela, Spain.

Application	Source	Description	Outcome
Apache1.1 Beta	Bug number 1507	AppenderAttachableImpl object should be protected by synchronization in AsyncAppender.getAllAppenders	Exception
MySQL6.0	Bug number 48930	lock_state is updated by two different threads holding different mutexes	System hangs
Mozilla-JS	Bug number 622691	The write to cx→runtime→defaultCompartmentIsLocked is not consistently protected by the lock	Incorrect result
Mozilla-XPConnect	Bug number 557586	One thread sets gLock to null before another thread drops the lock	Segmentation fault
Mozilla-Video/Audio	Bug number 639721	mInfo is written by nsBuiltinDecoderReader without its lock while mInfo is read from HaveNextFrameData with a lock	Incorrect result
Pbzip2-0.9.4	Paper [29, 32]	main() frees fifo→mut without protection	Segmentation fault
Windows Kernel	Case study 2 in slides of [8]	Two threads access the same structure with different mutexes	Incorrect result
Windows Kernel	Case study 3 in slides of [8]	parentFdoExt→idleState is not protected by a lock	Incorrect result
Windows Kernel	Real data race example in [8]	gReferenceCount is updated without protection	Incorrect result
Trie benchmark	An example in [22]	The prefix match function reads the leaf field of the root object without acquiring a lock on the trie	Incorrect result

Table 1. Real examples of harmful asymmetric data races. We found that 20% of harmful data races are asymmetric.

This paper is organized as follows: Section 2 motivates the problem; Sections 3, 4, and 5 describe Pacman's architecture and implementation; Section 6 evaluates Pacman; Section 7 discusses related work; and Section 8 concludes.

2. Asymmetric Races: Common & Harmful

The focus of this paper is a common and likely harmful type of data race called *Asymmetric*. This is a data race where at least one of the racing threads is inside a synchronization-protected critical section [22, 23]. In addition, we are interested in efficiently *tolerating* them in production runs.

Harmful asymmetric data races are common in the real world. To assess their frequency, we examined 50 harmful data race bugs from bug libraries of open source software and from Microsoft reports. We define harmful as being a bug that the user wants fixed — as opposed to the many data races explicitly created by the programmer for performance. Of the 50 harmful races, we found 10 that are asymmetric. This is a significant 20%. They are shown and described in Table 1.

The high frequency of asymmetric data races is confirmed by Microsoft researchers in [22, 23], who claim that they frequently encounter them in software development. They provide two intuitive sources of asymmetric data races. One source is code developed by good software developers that has to share memory state with less-tested code developed outside of the house — e.g., various device drivers. A second source is legacy. Specifically, a library may have been written assuming a single-threaded environment, but later the requirements change to multithreading. This requires that all the threads acquire a lock before accessing shared state. Unfortunately, some corner cases are missed.

Asymmetric data races are likely harmful. Indeed, all of the ones shown in Table 1 that come from bug libraries have been confirmed as bugs in the libraries, and fixed in future releases of the software. In addition, the fact that the programmer protected one thread's accesses to the racy variables in a critical section suggests that these are important variables. The atomicity of the critical section accesses, as intended by the programmer, is broken through accesses from other threads; this is likely to be harmful.

2.1. Our Goal

Our goal is to *tolerate* asymmetric data races in production runs without needing training tests. This approach is complementary to conventional in-house data-race debugging. It is motivated by four facts. First, even after extensive testing, date race bugs appear in released code. Second, it often takes years between the time when a bug is detected in the field and when a fix is available from the vendor [29]. Third, for the fraction of asymmetric races caused by third party or (perhaps) legacy code, fixing the bug may not be a feasible option because the source code may be unavailable. Finally, the structure of these races already suggests a way to minimize their potential harm: prevent the unsafe thread from corrupting the state or reading inconsistent state while the safe one is in the critical section.

3. Pacman: Tolerating Asymmetric Races

3.1. Overview of the Idea

We want to prevent unsafe threads from corrupting the state or reading inconsistent state while the safe thread is in the critical section. We must ensure that an access *A* from an unsafe thread that conflicts with an access inside the critical section is ordered in the same way with respect to all of the accesses in the critical section. As shown in Figure 2(a), the first write by T2 can proceed, but the second one has to be prevented until after the unlock. Similarly, the first read by T2 in Figure 2(b) can proceed, but the second one has to wait until after the unlock.

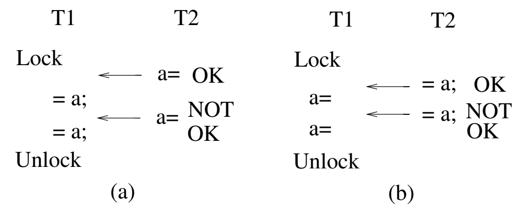
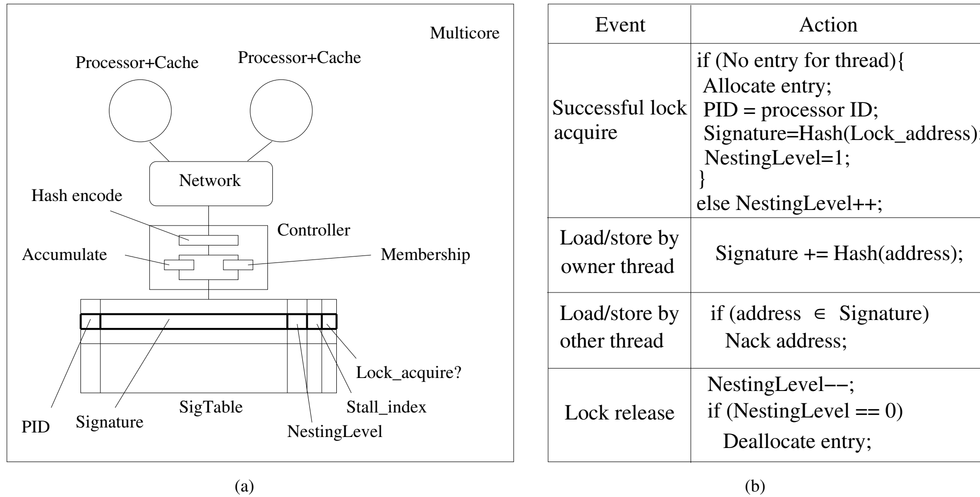


Figure 2. General approach to handle asymmetric data races.

The idea behind Pacman is to leverage the hardware cache coherence protocol in a multiprocessor to temporarily protect the variables that a thread is accessing in a critical section. The hardware performs two concurrent actions. One is to record the addresses of (a subset of) the variables that the safe thread is accessing while executing a critical section. In fact, to a large extent, we only need those addresses that can be observed by the cache coherence protocol, as we will see. The second action is to reject any requests from the unsafe threads that conflict with these variables, until the safe thread leaves the critical section.

For efficiency, Pacman does not record the addresses in a table. Instead, it uses a Bloom filter [2] to encode them into a hardware



Event	Action

Successful lock acquire	if (No entry for thread){
	Allocate entry;
	PID = processor ID;
	Signature=Hash(Lock_address);
	NestingLevel=1;
	}
	else NestingLevel++;

Load/store by owner thread	Signature += Hash(address);
Load/store by other thread	if (address ∈ Signature) Nack address;
Lock release	NestingLevel−−; if (NestingLevel == 0) Deallocate entry;

(b)

Figure 3. SigTable organization (a) and operation (b).

address signature. Moreover, to make the hardware as unintrusive as possible, the signature is stored in a module called *SigTable* that is connected to the on-chip network and sees all coherence transactions. Physically, the SigTable is associated with the bus controller in a bus-based multiprocessor, or is distributed across the different directory modules in a directory-based multiprocessor. Since multiple processors may be executing critical sections concurrently, the SigTable stores as many signatures as critical sections are in progress.

The application code is unmodified. However, Pacman assumes that the critical section entry and exit points of safe threads are marked in the code with synchronization macros or libraries. Inside these macros or libraries, Pacman makes sure that there is a network access, implemented as part of the synchronization operation as we will see. As a result, the SigTable always knows when a processor enters and exits a monitored critical section.

In this section, we describe Pacman's basic operation and the two key aspects that affect the ability to tolerate data races: caches and stalls.

3.2. Pacman's Basic Operation

The SigTable is a hardware table that stores the addresses accessed by each in-progress critical section, and prevents accesses by other processors to these addresses. In this discussion, we describe a centralized SigTable, as it would be used in a bus-based system; later, in Section 5.4, we outline a distributed one to be used in a directory-based system. Figure 3(a) shows the SigTable, which has one entry (a row) for each in-progress critical section. In each entry, the two main fields are *PID* and *Signature*. PID is the ID of the processor currently executing the critical section that owns the entry. In Section 5, we virtualize the SigTable. The Signature field contains (in an encoded form) the addresses of the lines accessed by the thread in the critical section so far and observed by the SigTable. A controller at the SigTable input takes the addresses of protocol transactions, hash-encodes them with a hardware Bloom filter [2], and may accumulate them into signatures and/or check them for membership in signatures [4].

The SigTable operates as follows. When a lock acquire successfully grabs a lock, the SigTable allocates a new entry for the critical section, sets PID to the requesting processor ID and, after clearing Signature, it inserts the hashed physical address of the lock in it. After this, at every load and store issued by the thread that is not intercepted by the cache, the SigTable hash-encodes the address of the line accessed and accumulates it in Signature. During this time, network accesses by other threads are hashed and checked for membership in Signature. If there is a match, the request is Nacked (negative-acknowledged) to the requester, which will retry. Finally, when the thread releases the lock, the SigTable deallocates the entry.

Pacman flattens nested critical sections, accumulating all the addresses accessed in the nest in the Signature. To support this feature, SigTable entries have a *NestingLevel* field. On a successful lock acquire, if the processor does not own a SigTable entry yet, the SigTable proceeds as above and sets NestingLevel to 1; otherwise it increments NestingLevel. On a lock release, the SigTable decrements NestingLevel and, if it is zero, deallocates the entry. Figure 3(b) lists the overall SigTable operation.

With this approach, Pacman isolates the critical section from unsafe threads. Note that Pacman needs no compiler support, no OS support, and no source code changes. Moreover, it has negligible execution overhead for the safe thread.

Nacks are often used in cache coherence protocols, to avoid having to buffer messages that cannot be processed immediately [9]. While they can cause traffic hot spots in pathological cases, the probability of an asymmetric race is low enough that there is no need to provide any contention management mechanism.

Finally, while Pacman has a transactional memory (TM) [10] flavor, it needs none of TM's key mechanisms such as speculation, rollback, timestamp support or contention management (Section 7.2).

3.3. Cache Effects

Since the SigTable is placed in the network, it does not see the accesses intercepted by the caches. To capture the required information to guarantee the atomicity of the critical sections, it relies only on the transactions induced by the cache coherence protocol — plus some small extensions that we will explain. We now show why this is the case. In the following discussion, we assume a basic MESI cache coherence protocol. Other protocols may require slightly different considerations.

Figure 4 shows two simple patterns. In Figure 4(a), thread T1 writes to line *x* and misses in the cache. SigTable records the address. Any subsequent read or write to *x* by T2 requires a coherence transaction, which is observed and Nacked by the SigTable. In Figure 4(b), T1 reads *x* and misses in the cache. SigTable records the address. If T2 reads *x*, there may or may not be a coherence transaction. If there is, the access will be Nacked; otherwise, it will not. Either situation is fine because two reads do not conflict. However, if T2 writes *x*, there is a coherence transaction that will be Nacked by the SigTable.

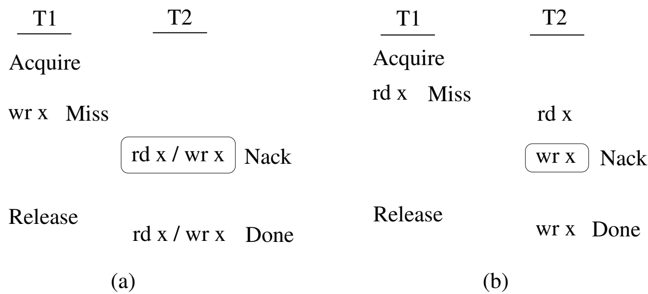


Figure 4. Examples to help understand Pacman's operation.

The more involved cases involve three issues: cache state before entering the critical section, cache displacements during the critical section, and synchronization operations. We consider each in turn.

3.3.1. Cache State Prior to Entering the Critical Section

Before the safe thread enters the critical section, some of the lines in its cache may be in a state that enables the processor to access them silently during the critical section. There are two cases: when *x* was Dirty (or Exclusive) in T1's cache in Figures 4(a) and (b), and when *x* was Shared in T1's cache in Figure 4(b). In these cases, SigTable will not observe T1's access to *x*.

None of the two cases prevents SigTable from ensuring the atomicity of the critical section. Consider the case when *x* was Dirty (or Exclusive) in T1. When T2 attempts to access the line and misses, the coherence protocol forces T1 to write back the line. When the SigTable sees that a processor with a SigTable entry writes back a line, it assumes that the processor had accessed the line. Consequently, while allowing the line to be written to memory, it inserts the line's address in the entry's Signature and Nacks the requesting (unsafe) processor — hence ensuring critical section atomicity. No functional change to the caches or coherence protocol is needed. If T1 had not accessed the data in the critical section, Pacman acts conservatively but not incorrectly.

Consider now the case when *x* is Shared in T1. When T2 attempts to write the line, the hardware issues a coherence transaction that invalidates T1's copy. For this case, Pacman requires a simple hardware extension. Specifically, it requires that T1's cache informs, in its response to the invalidation, that indeed, it has invalidated a line. When the SigTable sees that a processor with a SigTable entry has invalidated a line, it assumes that the processor had accessed the line. Consequently, it inserts the line's address in the Signature for the entry and Nacks the requesting (unsafe) processor. Again, if T1 had not read the line in the critical section, Pacman acts conservatively but correctly.[1]

[1] In all of these cases, a Nacked write has already invalidated the line from all the caches. This can hurt performance slightly if caches have to re-access the data. However, this occurs only once.

Supporting this change is simple. In a directory-based protocol, when a cache invalidates a line, it must set a bit in the invalidation acknowledgment returned to the directory. In a snoopy-based protocol, the cache must set a bit in the bus that is visible to the SigTable. This hardware change and all the other processor/cache modifications required by Pacman will be summarized in Section 3.3.4.

3.3.2. Cache Displacements During the Critical Section

Consider the case when, as a processor executes a critical section, its cache displaces a line that was in the cache before the processor entered the critical section. Such line is not in the Signature, but must be conservatively put there as the processor may have accessed it silently during the critical section.

There are two cases, namely that the displaced line is Dirty or not. If it is, the case is easy: as the line is automatically written back to memory, the SigTable sees that the source processor owns an existing SigTable entry and inserts the address in the Signature.

If the line is not Dirty, the coherence protocol would not trigger a line writeback. Therefore, we propose to modify the cache controller to send a notification to the network when the cache displaces a clean line inside a (monitored) critical section. The notification carries the address of the line. When the SigTable sees such a notification from a processor that owns a SigTable entry, it conservatively accumulates the address in the Signature. The extra traffic created is small, since critical sections are typically short. Overall, this extension is like the Replacement Hint transaction sometimes used in directory protocols [5], except that it only needs to occur while the processor is inside a critical section.

This is the most significant hardware modification required by Pacman, as summarized in Section 3.3.4. However, it can be implemented easily. Specifically, the controller for the last level of private cache has a counter register called *Mode*. When Mode is not zero, the cache is in *Notification* mode, and it sends a notification message at every displacement of a non-Dirty line. Every successful lock acquire operation for a monitored critical section increments the Mode register, while every release for it decrements it. This ensures that, in nested critical sections, the cache remains in Notification mode throughout the outermost critical section. Increments and decrements can be supported with a write to a register in the cache controller. Such writes can be performed inside acquire and release macros or libraries, such as those of M4 [13] or OpenMP [6].

3.3.3. Synchronization Operations

The SigTable must see all of the successful acquires and all of the releases. This is because they may allocate/deallocate a SigTable entry and update the Signature and NestingLevel fields. The coherence protocol ensures that the SigTable sees these synchronization operations except in the cases when they hit a cache line in state Dirty or Exclusive. So, we must ensure that, in these cases, a notification access is also issued to the network that the SigTable sees.

To accomplish it, we propose to augment the implementation of the acquire and release instructions. If a successful acquire or a release operation proceeds *without* needing a network access, the hardware issues a notification message to the network. An alternative design would involve not changing the acquire or release instructions and adding an explicit uncached write inside the synchronization macros or libraries. While this design is simpler, it would add more overhead to the synchronization operation. Still,

overheads may be tolerable, especially if one is willing to identify the potentially problematic critical sections and only monitor those.

Unsuccessful acquires do not need to be observed by the SigTable.

Pacman is compatible with modern processors that speculatively read past an acquire before the acquire completes. The SigTable may be unallocated and, therefore, unable to capture the loaded address. The effect is the same as if the load had hit in the cache (Section 3.3.1).

3.3.4. Summary of Cache Hierarchy Modifications

Table 2 summarizes the functional modifications that Pacman requires in the cache hierarchy and coherence protocol. We believe that these modifications are modest. All of the other modifications are unintrusive because they are part of the SigTable module.

When a cache invalidates a clean line, it sets a bit that is visible to SigTable.
In Notification mode, the last-level private cache sends a notification message when it displaces a clean line.
A successful acquire or a release that are fully intercepted by the cache issue a notification message to the network.

Table 2. Pacman functional modifications in the cache hierarchy and coherence protocol.

3.4. Multiple Stalls and Deadlock

Pacman temporarily stalls unsafe threads by Nacking their conflicting requests. We now consider the case when multiple threads are Nacked and show how deadlock can occur and is handled.

3.4.1. Multiple Thread Stalls

It is possible that two (or more) threads send Nacks to each other and end up all stalling. This situation can occur due to three reasons: some race bugs where all of the threads synchronize, false sharing and false positives. Figure 5 shows two examples of the first case. In Figure 5(a), two threads T0 and T1 acquire two different locks L0 and L1, respectively. Inside the critical sections, both threads access the same two variables g0 and g1 in different order. The timing is so unfortunate that each thread accesses one variable and then receives a Nack on attempting to access the second variable. We have formed a cross-thread stall cycle and no thread can make progress.

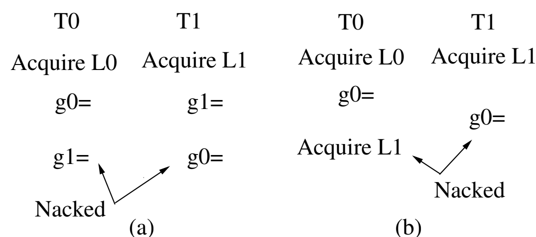


Figure 5. Examples of data race bugs where all the threads synchronize and lead to deadlock.

In Figure 5(b), the two threads T0 and T1 acquire two different locks L0 and L1, respectively, and then access the same variable g0. T0 succeeds and T1 gets Nacked. Then, the thread that succeeds (T0) attempts to acquire the lock of the other, stalled thread (T1). We have a cross-thread stall cycle as before, except that one of the two dependences in the cycle is for a lock variable.

The second source of cross-thread cycles leading to deadlock is pathological cases of false sharing of lines. For example, it occurs in a pattern similar to Figure 5(b) except that T1, rather than accessing variable g0, accesses a variable that shares a line with g0. Recall that the SigTable is in the network and can only see line addresses.

The third source of cross-thread cycles leading to deadlock is pathological cases of false-positive dependences between threads, due aliasing in the signatures or due to the cache state prior to entering the critical section (Sections 3.3.1 and 3.3.2). However, since critical sections tend to be small, false positives are typically not very significant.

We will see in Section 7.1 that one of the existing software proposals for tolerating asymmetric races called ISOLATOR [22] also suffers from the first two sources of deadlock. In fact, since ISOLATOR's protection granularity is a page (rather than a cache line as in Pacman), it is very vulnerable to false sharing. In ISOLATOR, when such cycles occur, threads keep retrying, until the software detects that a certain time has elapsed without making forward progress. Execution is then interrupted. Unfortunately, such a timeout-based approach to detect deadlocks is very slow.

3.4.2. Making Forward Progress in Pacman

Pacman uses *hardware* to detect a deadlock cycle *as soon as* the memory access that closes the cycle occurs. This approach is much faster than the software-based timeout approach of ISOLATOR. Moreover, at that point, Pacman's hardware breaks the cycle by allowing one the stalled threads to perform one memory access. Such access enables forward progress.

To support the algorithm, we add two fields to each row of the SigTable (Figure 3(a)). First, *Stall_index* tells if the thread that owns the entry is being Nacked. Specifically, *Stall_index* stores the index of the SigTable entry that sends Nacks to the owner thread. If the owner thread is not being Nacked, this field is null. Second, when *Stall_index* is not null, the *Lock_acquire?* bit is set if the owner thread is being Nacked while trying to acquire a lock. This bit will detect the case of Thread T0 in Figure 5(b).

When an access by processor P_i is Nacked by entry *j* of the SigTable, the SigTable hardware checks if P_i also has an entry in the SigTable. If so, it sets that entry's *Stall_index* to *j* and, if appropriate, sets the *Lock_acquire?* bit. Then, the SigTable hardware follows the *Stall_index* pointer by checking entry *j* in the SigTable and reading its own *Stall_index*. If, by following the *Stall_index* pointers in this way, the hardware ends up in entry *i*, it has detected a cycle. At that point, the hardware needs to decide which thread among those in the cycle is allowed to perform one access without being Nacked. A simple approach is to pick one of the threads that holds locks requested by other threads (such as T1 in Figure 5(b)). Such threads are detected from the *Lock_acquire?* bit of other entries, and they need to make progress to break the cycle. If there is no such thread, the hardware picks one thread at random. The next time that the SigTable sees a request from the picked thread, it does not Nack it.

3.4.3. Breaking the Atomicity of Critical Sections

With the algorithm described, Pacman immediately finds and breaks any deadlock — unless it was already present in the original application. However, by letting one stalled thread complete one access, it can conceivably break the atomicity of a critical section. To understand the problem, we consider each of the three sources of deadlock listed in Section 3.4.1.

In the first case (some race bugs where all of the threads synchronize), Pacman can potentially break the atomicity of one of the critical sections. While Pacman could be designed to break only the atomicity of unsafe threads, such an approach would not work for all the race bugs. An example is when T1 in Figure 5(b) is the unsafe thread. Overall, given the very low probability of breaking atomicity in this way, we do not attempt to avoid it.

In the third case (false positives), letting one thread proceed does not break the atomicity of any critical section.

In the second case (false sharing), atomicity can potentially be broken unless special care is taken. To see why, consider Figure 6, which is slightly modified over Figure 5(a). In this example, variables g0 and g0' share the same cache line, while g1 and g1' share another line. Because of false sharing, threads T0 and T1 deadlock. By breaking the deadlock through letting T1 read g0', Pacman is allowing the line to go to T1's cache. Right after the critical section, T1 could attempt to silently access g0 from its cache, which could break T0's atomicity.

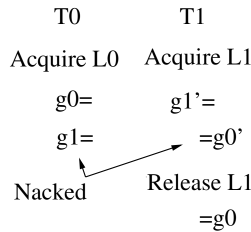


Figure 6. Atomicity could be broken due to false sharing.

To prevent this case from occurring, we could augment Pacman so that, when the SigTable lets one access break a deadlock, it marks it as non-cacheable. The requesting processor would be allowed to use (read or write) the word, but its cache would not be allowed to keep the line. As a result, accesses to other words would miss in the cache. This extension would avoid breaking atomicity when false sharing occurs between words. However, a more elaborate solution would be needed when false sharing occurs between bytes of the same word. Given the very low probability of breaking atomicity due to false sharing, Pacman does not include this support.

4. Discussion

Pacman's unique goal makes it very different from hardware-based race detectors [14, 16, 19, 20, 34]. In these schemes, the goal is to characterize and debug races. Moreover, false positives are highly undesirable. Hence, these schemes tend to use more expensive hardware, such as per-word access information, epoch IDs in coherence messages, and even rollback. Pacman's goal is to *tolerate* asymmetric races in production runs. Since we are not debugging, it is fine to have some false positives (e.g., due to aliasing in signatures) if they are handled fast. A false positive in Pacman simply slows down a thread a little bit. The result is cheaper, less intrusive hardware. Still, Pacman could be used as a detection tool for asymmetric races. Indeed, the number of false positives we found is very low (as we show in the evaluation) and the number of false negatives is likely negligible (as we summarize in Section 7.1.1).

Pacman provides a powerful primitive: dynamically and selectively prevent accesses to a set of addresses by certain processors. It can be used in security and performance/correctness debugging. For example, it can enforce atomic regions and detect atomicity violations, or provide watchpoint capability.

Pacman is fastest when critical sections are small, which is the norm in many codes. However, we believe that it is also very useful for beginner programmers, who tend to write long critical sections. The long critical sections will be protected and the program will run safely, although slower.

It is possible that a malicious thread could attempt to use Pacman to deny access to other threads, by remaining inside a critical section and filling up a signature. This problem can be detected with a watchdog timer, or by counting the number of Nacks triggered by a critical section.

Pacman has a few limitations. One is that it needs to be able to identify (monitored) critical section entry and exit points. To do so, we have assumed synchronization macros or libraries, but certain types of code are not written in this way. Second, the fact that all of the successful acquires and releases need to access the SigTable can slow down codes where the same thread repeatedly executes the same, short critical section. We have not seen this case but it is possible.

A final limitation is that Pacman is not designed for some unusual types of critical sections. They include million-instruction critical sections. They also include patterns where a thread spins on a flag inside a critical section, waiting for a racy thread to set the flag (Figure 7). We feel that this pattern is bad programming style. In any case, for these types of critical sections, the compiler or programmer can disable Pacman or use plain synchronization. Alternatively, Pacman can have a watchdog timer or a Nack-counting mechanism that detects the problem and allows the write(s).

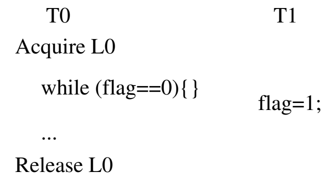


Figure 7. Unusual pattern that Pacman does not handle.

5. Implementation Issues

5.1. Pacman Module

The Pacman module is a hardware module connected to the on-chip network (Figure 8(a)). It comprises the SigTable and its controller. The controller is composed of two simple hash blocks (*H-Blocks*) and the Cycle Detection & Breakup module. The latter chases the *Stall_index* links as described in Section 3.4.2 to detect and break deadlocks.

Figure 8(b) shows H-Block$_1$ and the SigTable. H-Block$_1$ takes the address of an incoming request transaction and encodes it into a signature using a parallel Bloom filter [2] ($Signature_{in}$ in Figure 8(b)). The signature is then tested for membership in valid SigTable entries from other processors ($\in$ in Figure 8(b)). This operation involves a bit-wise AND operation to get the intersection and then a check for zero [4] (Figure 8(c)). If any membership test is positive, the Nack$_1$ signal is raised. Otherwise, if the requesting processor owns a SigTable entry (or a new one needs to be allocated), the signature is bit-wise ORed with the correct SigTable entry ($\cup$ in Figure 8(b) and expanded in Figure 8(d)). Overall, H-Block$_1$'s operations can be performed in 2-3 cycles and are hidden under the first half of the bus transaction.

In the second half of the bus transaction, when the caches have finished snooping, the bus may receive a write back or invalidation

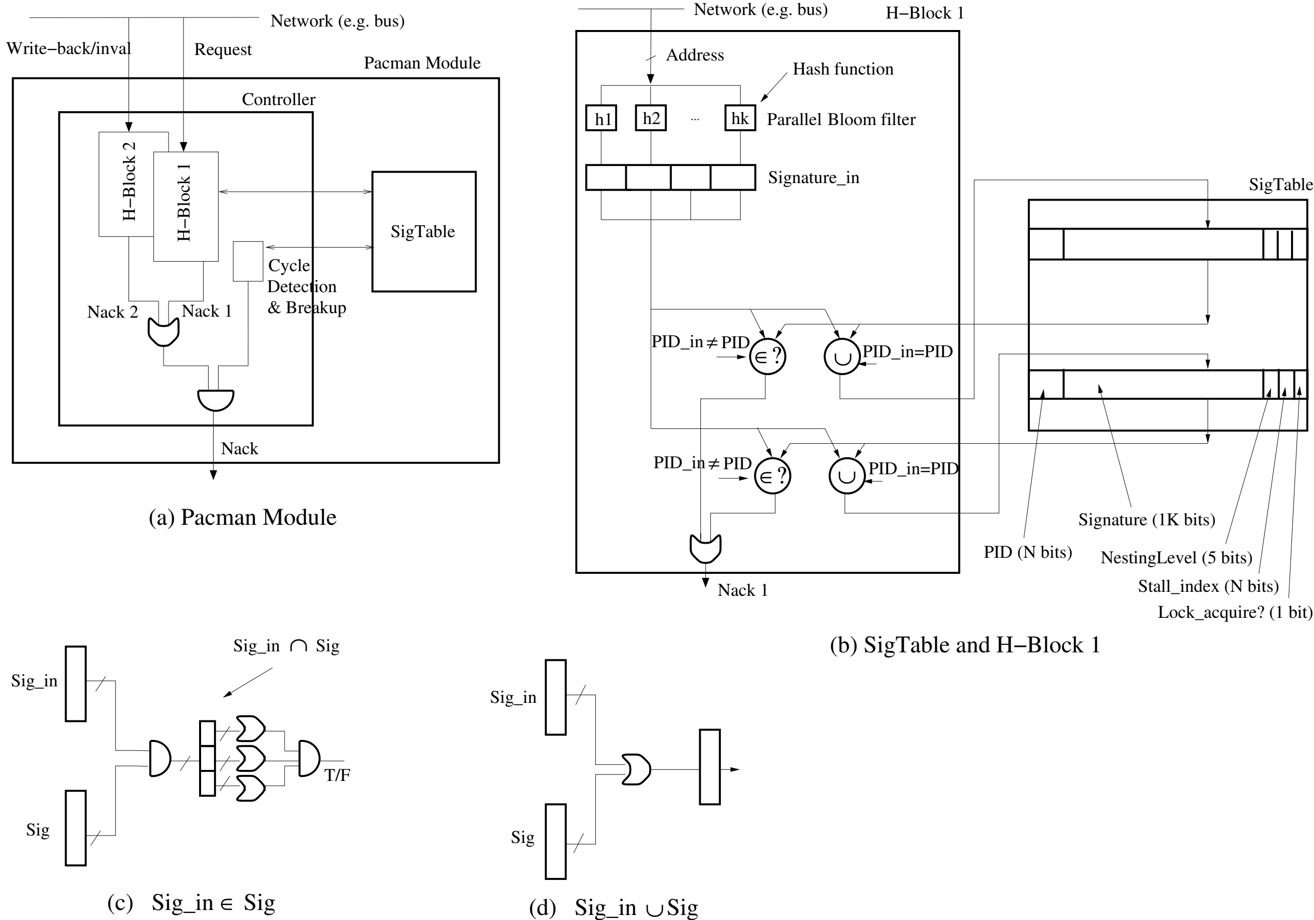


Figure 8. Implementation of the Pacman module.

response (Section 3.3.1). H-Block$_2$ (not shown in detail) checks if the processor ID that writes back or is invalidated has a SigTable entry. If so, it bit-wise ORs the hashed address with the correct signature and raises Nack$_2$. H-Block$_2$'s operation takes 1-2 cycles. If Nack$_1$ or Nack$_2$ is raised and the Cycle Detection and Breakup module does not prevent it, a Nack signal is returned on the bus.

All of the operations of the Pacman module except for cycle detection are simple enough to be overlapped with the bus transaction. In a directory protocol, they overlap with directory module accesses. The cycle detection may take over 10 cycles, which is acceptable since it is done in the background.

Figure 8(b) also shows the sizes of the SigTable's fields. The size of PID and Stall_index depend on how many threads we may need to monitor at a time. For Signature, we found that, with 1,024 bits, false positives are typically less than 1%. For NestingLevel, we allocate 5 bits, which is enough for our programs.

Finally, the Pacman module is enabled and disabled by the *Pacman_on* and *Pacman_off* commands, respectively. They can be implemented as writes to memory-mapped registers. These commands can be used to exclude the program regions that are serial or otherwise uninteresting.

5.2. Virtualization: Thread Pre-emption and Migration during Critical Section Execution

While executing a critical section, a thread can be pre-empted and even migrated to another processor. In an advanced design that requires OS support, we would like that (i) while a thread is pre-empted in a critical section, we keep protecting its critical section, and (ii) when it resumes in a potentially different processor, we keep accumulating its accesses in the same signature. To support this, when the OS pre-empts a thread from processor *i*, it checks the SigTable for an entry with PID equal to *i*. If it finds one, it changes its PID field. Specifically, if the thread will not run anywhere, it sets the PID field to a special code (e.g., *OUT*); if it will run on processor *j*, it sets the PID field to *j*.

With this algorithm, if a thread gets pre-empted and is not running, it still has its critical section protected from asymmetric races. Indeed, its SigTable entry is still valid and coherence messages are checked against its signature. The checks may result in sending Nacks. Then, when the thread is scheduled on a different processor, its accesses are still accumulated into the same old signature.

This approach is efficient, since there is no copying or saving/restoring of SigTable entries. Moreover, the hardware is kept simple, since it always does the same thing: accumulate accesses from processor *i* into the SigTable entry tagged with PID *i*. Stall_index does not get stale, since it contains a table index.

If the program has more threads than processors, there may be several SigTable entries with a PID equal to OUT. In addition, at a given time, the SigTable entries may belong to threads from several *different programs*. Pacman works correctly because it uses physical addresses.

There is an issue with the cache state left behind by a thread that migrates while executing a critical section. Recall from Section 3.3.1 that the thread may have entered the critical section with

Architecture	CMP with 4 or 8 processors	Coherence protocol	Snoopy basic-MESI on 64byte bus
Processor type	2-issue, in-order, 1GHz	SigTable parameters	From Figure 8. Max: 8 rows
Private L1 cache	32Kbytes, 4-way asso., 64byte lines	Signature size	1,024 bits
Private L2 cache	512Kbytes, 8-way assoc., 64byte lines	Signature structure	8 128-bit Bloom filters with H3
L1 hit latency	2 cycles round trip	Cycle detection latency	4-14 cycles
L2 hit latency	8 cycles round trip	H-Block$_1$ latency	2 cycles
L2 miss latency	30 cycles round trip to other L2s	H-Block$_2$ latency	2 cycles
L2 miss latency	250 cycles round trip to memory		

Table 3. Default architectural parameters.

cache state that it later accessed while in the critical section without notifying the SigTable. We showed that Pacman (conservatively) captures this information at cache displacements or at writebacks/invalidations triggered by other processors. However, if we now migrate the thread, we cannot capture such events.

To keep the design simple, we accept this limitation. This means that Pacman misses the few cases listed in Sections 3.3.1 and 3.3.2 for threads that migrate while in a critical section. A more aggressive approach would be to write back to memory all the dirty cache lines at the time the thread migrates while in a critical section. The addresses of these writebacks would be put in the signature. A more drastic approach would be not to allow migration during critical section execution. Overall, since critical sections are typically small, migration during their execution is rare and does not justify additional actions. Like all data-race handling techniques, Pacman is a best-effort approach.

5.3. Extensions for Multithreaded Processors

Multithreaded processors have multiple hardware contexts and run multiple threads at a time. It is possible that different threads executing on different contexts of the same processor concurrently execute different critical sections. In this environment, Pacman requires an extension where the messages sent by processors to the SigTable include both the processor ID and the hardware context ID within the processor. Similarly, SigTable entries have both a PID and a ContextID field.

The cache-state issues of Sections 3.3.1 and 3.3.2 are handled conservatively. If multiple contexts in a processor are concurrently executing critical sections, any writeback, invalidation, or displacement that needs to insert an address in a signature, does insert it in all the SigTable entries owned by that processor.

Since the SigTable is connected to the network, it can only observe data sharing across processors, not across contexts in a processor. Consequently, for Pacman to tolerate races as advertised, a program can only use one context per processor — although multiple programs can use the multiple contexts of a processor. To allow a program to use multiple contexts in a processor, bigger changes would be needed, such as stalling all the other threads in the processor when one thread is executing a critical section.

5.4. Extensions for a Distributed SigTable

The discussion so far assumed a centralized SigTable, which is reasonable for a snoopy protocol. To use Pacman in a system with a directory-based protocol, we need to distribute the SigTable across the different directory modules. Since such a design is outside our scope, we only outline it.

Like the directory, the SigTable naturally lends itself to a distributed environment, with partitions based on address ranges. Consequently, each directory module has an associated SigTable module, which is in charge of the range of physical addresses assigned to the local directory module. When a thread enters a critical section, the hardware allocates an entry for the processor in all the SigTable modules; when it exits it, all the entries are deallocated. When a thread misses on an address, the request naturally reaches the home directory of that address. There, the address is checked against the entries in the local SigTable module using the usual algorithm. The SigTable modules in the other directory modules are not checked.

6. Evaluation

6.1. Experimental Setup

To evaluate Pacman, we instrument parallel application binaries with Intel's Pin framework connected to a cycle-by-cycle execution-driven architecture simulator based on SESC [24]. The simulator models a chip multiprocessor (CMP) with 4 or 8 processors. The default parameters of the architecture are shown in Table 3. The processors are two-issue, in-order, and overlap memory accesses with instruction execution. Each processor has a private cache hierarchy kept coherent by a basic MESI coherence protocol on an on-chip bus. The bus is connected to the SigTable and to off-chip main memory. Unless otherwise indicated, the sizes of the fields in a SigTable entry are those shown in Figure 8. To generate a signature, Pacman uses 8 128-bit Bloom filters in parallel using the H3 hash function from [25], for a total of 1,024 bits per signature.

For sensitivity analysis, we consider two cache hierarchy models, namely one where each processor only has an L1 cache, and one where it has both a private L1 and a private L2. The first model puts more pressure on Pacman.

We evaluate Pacman with all the 14 SPLASH-2 applications, the 12 PARSEC applications that support pthreads, the Sphinx3 speech recognition software [26], and Apache-2.2.3. The SPLASH-2 codes use their default inputs, while the PARSEC ones use the *simmedium* inputs. For Sphinx3, we use the test input provided, which executes over 500 million instructions, while for Apache, we set up clients that keep sending requests to the server, so that the server executes around 40 million instructions.

In our evaluation, we slightly modify the Canneal and Ferret applications. At the beginning of Canneal, a thread uses a critical section to initialize a large memory space — even though there is no other active thread at that time. Consequently, we turn off Pacman during that time. In Ferret, each thread initializes a random number generator within a critical section. Since only the seed is a shared variable, we move the local-variable accesses in the random number generator initialization routine outside of the critical section. If we did not do these changes, the statistics on critical section sizes (Section 6.2) would be biased. In addition, for Ferret, if we inserted all the local-variable addresses into the signature, we could potentially induce, through address aliasing in the signatures, false positive conflicts with other threads, and unnecessarily stall them.

In the rest of this section, we characterize the critical sections, evaluate the overheads of Pacman, and examine the asymmetric data races discovered by Pacman.

Category	Application	# Dynamic CS	CS Insts (%)	#Insts per CS	Max #Insts in CS	#Rd per CS	#Wr per CS	#Clean disps per CS	#Sig addrs per CS	Max # sig addrs in CS	Max CS nesting level
SPLASH-2 Kernels	cholesky	6,957	0.0	30.3	161	10.7	4.7	0.0	6.4	11	1
	fft	32	0.3	33.9	47	11.9	10.5	0.1	5.8	7	1
	lu/contiguous	272	0.0	36.1	47	12.6	10.7	0.0	6.0	7	1
	lu/non_cont.	80	0.0	35.2	47	12.4	10.5	0.1	5.8	8	1
	radix	78	0.0	26.1	47	9.4	8.4	0.0	5.9	7	1
SPLASH-2 Apps	barnes	68,938	0.4	118.1	1,898	40.1	29.3	0.0	11.9	56	1
	fmm	44,622	0.2	142.1	252	54.7	27.9	0.0	13.4	21	1
	ocean/cont.	4,432	0.0	31.5	45	11.8	9.6	0.0	6.8	9	1
	ocean/non_cont.	4,312	0.0	30.9	45	11.8	9.5	0.0	5.9	7	1
	radiosity	273,087	0.9	18.2	1,226	8.7	5.9	0.0	5.6	89	5
	raytrace	95,475	0.3	29.3	6,661	7.5	5.8	0.0	6.0	343	1
	volrend	72,524	0.0	12.1	50	5.0	3.0	0.0	4.9	8	1
	water-nsquared	6,292	0.0	50.3	51	34.4	12.4	0.0	17.0	18	1
	water-spatial	157	0.0	23.8	47	9.6	7.0	0.0	6.0	9	1
PARSEC Kernels	canneal	4	0.0	7.0	10	2.5	3.5	0.0	3.3	4	1
	dedup	17,932	0.1	315.9	802	121.2	67.9	0.1	14.4	33	1
	streamcluster	52,128	0.0	21.0	32	7.1	4.8	0.0	3.2	5	1
PARSEC Apps	blackscholes	0	–	–	–	–	–	–	–	–	–
	bodytrack	8,273	0.0	37.0	1,228	15.6	11.1	0.0	6.9	34	1
	facesim	7,921	0.0	37.0	154	18.0	9.9	0.0	5.4	11	2
	ferret	733	0.0	19.2	44	5.4	7.2	0.0	5.0	9	2
	fluidanimate	2,113,870	0.7	15.9	32	10.2	4.1	0.0	8.0	10	1
	raytrace	73	0.0	7.8	31	2.6	2.3	0.0	2.2	6	1
	swaptions	0	–	–	–	–	–	–	–	–	–
	vips	14,056	0.0	49.0	6,723	18.6	11.8	0.0	8.0	106	23
	x264	4,071	0.0	10.6	39	5.9	1.7	0.0	4.0	6	1
Other Apps	Apache	8,301	0.4	24.4	40	9.7	5.3	0.0	5.6	8	1
	Sphinx3	94,382	3.5	208.5	2,946	86.7	29.1	0.1	6.0	243	2

Table 4. Characteristics of the critical sections (CS) in the applications.

6.2. Characterization of the Critical Sections

Table 4 characterizes the critical sections in all 28 applications on the 4-processor CMP. Column 3 lists the number of dynamic critical sections in each program. Column 4 shows the percentage of the dynamic instructions in the programs that are inside the critical sections. We see that all programs but Sphinx3 execute less than 1% of their instructions in critical sections. The percentage in Sphinx3 is 3.5%. Columns 5 and 6 show the average and maximum number, respectively, of instructions executed per critical section. We see that the applications tend to have modest-sized critical sections. Most applications execute less than 100 instructions per critical section on average. The maximum number of instructions in a critical section reaches nearly 7,000 in Vips. Columns 7-8 list the average number of reads and writes per critical section.

Columns 9-11 correspond to the architecture with only the L1 caches. They show, per critical section, the average number of clean line displacements, and the average and maximum number of *line* addresses included in the signature. We can see that the average number of clean displacements per critical section is close to zero. This means that this effect is minor. The average number of line addresses included in a signature per critical section is typically less than 10 and, except for a few cases, the maximum number is not much higher. These numbers suggest that the probability of false positives in the signatures is low. Note that for the machine with both L1 and L2 caches, these numbers will be smaller. This is because caches keep more state.

The last column shows the maximum nesting level of critical sections. A value more than one means that the application has nested locks. We can see that most applications have a value of one. Only Radiosity and Vips, which have a recursive structure, have significantly deeper levels.

Overall, given the typical sizes and properties of the critical sections observed, we believe that a simple solution for asymmetric race detection is enough. Pacman provides such a simple solution.

6.3. Overheads of Pacman

There are two sources of execution overhead in Pacman. The first one is that some processors receive Nacks and have to retry. The second one is additional network traffic created by three event types: a notification message in a clean displacement inside a critical section, a retry after a Nack, and the extra message in a successful lock acquire or release that hits on a cache line that is in Dirty or Exclusive state.

Table 5 quantifies these effects for each application. Columns 3-8 show the total number of Nacks observed during the execution of the application. For each application, we performed 3-5 runs, and show the *maximum* number of Nacks seen in any individual run. The data corresponds to the architecture with L1 caches only, which is the worst case. Columns 3-5 correspond to 4-processor runs, while Columns 6-8 correspond to 8-processor runs. For Apache, since the server automatically sets the number of threads to a number larger than 8, we put the data under the 8-thread columns. In each group of three columns, the first one shows the Nacks observed due to true conflicts (i.e., two threads access the same variable), the second one the Nacks due to true conflicts or false sharing, and the last one the Nacks due to true conflicts, false sharing, or false positives.

The number of Nacks is very small. Only FMM and Bodytrack exhibit Nacks due to true conflicts. Each of them has one Nack. False sharing and false positives increase the number of Nacks. The highest number is 32 for Radiosity. This is negligible compared to the 454M dynamic instructions executed by Radiosity. Overall, the impact of any processor stall due to Nacks is negligible.

Columns 9-10 show the percentage increase in the network traffic due to the three effects listed above. Column 9 applies to the architecture with L1 caches only, while Column 10 applies to the one with L1 and L2. The data shows that the increase in traffic is very small. In the worst application, the increase is 1.5% for the case of L1 caches and 2.4% for the case of L1 and L2 caches. These

Category	Application	Number of Nacks (L1 only, 4 threads) True	True+FS	True+FS+FP	Number of Nacks (L1 only, 8 threads) True	True+FS	True+FS+FP	Increase in traffic with L1 only (%)	Increase in traffic with L1+L2 (%)	Sync hits per dyn inst (%)
SPLASH-2 Kernels	cholesky	0	0	0	0	0	0	0.0	0.0	0.00
	fft	0	0	0	0	0	0	0.0	0.0	0.00
	lu/contiguous	0	0	0	0	0	0	0.0	0.0	0.00
	lu/non_cont.	0	0	0	0	0	0	0.0	0.0	0.00
	radix	0	0	0	0	0	0	0.0	0.0	0.00
SPLASH-2 Apps	barnes	0	2	4	0	2	4	0.0	0.3	0.01
	fmm	1	1	1	1	1	1	0.0	0.1	0.00
	ocean/contiguous	0	0	0	0	0	0	0.0	0.0	0.00
	ocean/non_cont.	0	0	0	0	0	0	0.0	0.0	0.00
	radiosity	0	13	15	0	28	32	1.0	1.4	0.04
	raytrace	0	0	4	0	0	6	0.0	0.1	0.01
	volrend	0	0	0	0	0	0	0.0	0.1	0.00
	water-nsquared	0	0	0	0	0	0	0.0	0.1	0.00
	water-spatial	0	0	0	0	0	0	0.0	0.0	0.00
PARSEC Kernels	canneal	0	0	0	0	0	0	0.0	0.0	0.00
	dedup	0	0	0	0	2	2	0.1	0.2	0.00
	streamcluster	0	0	0	0	0	0	0.0	0.0	0.00
PARSEC Apps	blackscholes	0	0	0	0	0	0	0.0	0.0	-
	bodytrack	1	1	2	1	1	2	0.0	0.0	0.00
	facesim	0	0	0	0	0	0	0.0	0.0	0.00
	ferret	0	0	0	0	0	0	0.0	0.0	0.00
	fluidanimate	0	0	0	0	0	0	1.5	2.4	0.05
	raytrace	0	0	0	0	0	0	0.0	0.0	0.00
	swaptions	0	0	0	0	0	0	0.0	0.0	-
	vips	0	0	2	0	0	3	0.0	0.0	0.00
	x264	0	0	0	0	0	0	0.0	0.0	0.00
Other Apps	Apache	-	-	-	0	3	8	0.3	0.5	0.02
	Sphinx3	0	4	6	0	10	14	0.8	1.1	0.02

Table 5. Quantifying the sources of overhead in Pacman.

low numbers result from the fact that critical sections have a modest size and account for a small fraction of the execution time. Overall, the impact of this extra traffic is negligible.

Column 11 shows the number of successful lock acquires and releases that hit on a cache line that is in Dirty or Exclusive state and, therefore, introduce an additional bus access. The data corresponds to the architecture with both L1 and L2 caches. The column gives the number of such events as a percentage of dynamic instructions. We can see that, typically, such number is negligible. In the worst application, we have 0.05 such events per 100 instructions. Therefore, the impact of such events is negligible.

Finally, Figure 9 shows the increase in the execution time of the applications due to all of the Pacman overheads combined. The data is shown as a percentage of the original execution time of the applications and is plotted for 1, 4 and 8 threads. There is a data point for each program, and a line for the average of them all. The figure shows that, even for 8 threads, the maximum overhead in any application is only 0.4%, while the average is only 0.07%. The figure also shows that, for most applications, the overhead increases slowly with the number of threads. The overhead for 1 thread is due to the extra bus accesses in synchronizations. Overall, the execution time overhead of Pacman is negligible.

6.4. Unreported Asymmetric Data Race Bugs

Although the SPLASH-2 and PARSEC codes are widely used, Column 3 of Table 5 shows that Pacman discovered two true asymmetric data races: one in FMM and one in Bodytrack.

The asymmetric race in FMM is shown in Figure 10. It happens in subroutine *ComputeSubTreeCosts*, where multiple threads are accessing a tree structure. When two threads T1 and T2 are concurrently executing the subroutine, it may be that the two point to the same node from two different places (*pb* in T1 is the same as *b* in T2), and an asymmetric data race can happen.

The asymmetric race in Bodytrack happens between subroutine *Condition::Wait*, where variable *nWakeupTickets* is read and written inside a critical section, and subroutine *Condition::NotifyOne*, where it is written outside any critical section.

7. Related Work

7.1. Software Proposals for Asymmetric Races

To put our work in perspective, we describe in detail the two existing proposals to tolerate asymmetric data races, namely, ToleRace [23] and ISOLATOR [22]. Both schemes are software-only (i.e., they do not add any additional hardware). We then summarize Pacman's advantages over them.

In ToleRace, when a safe thread T_s enters a critical section, it makes two copies in software of all the shared variables in the critical section. Let us call the original variables *V* and the two copies *V'* and *V"*. The safe thread then executes the critical section reading and writing *V'*. In the meantime, any unsafe thread T_u can access the original variables *V*. When T_s completes the critical section, it compares *V* and *V"*. Based on whether *V* and *V"* are the same *and* on a knowledge of the access pattern interleaving of T_s and T_u, the safe thread makes one of three choices: (i) when T_u's execution can be serialized before T_s's, it copies in software *V'* to *V*, (ii) when T_s's execution can be serialized before T_u's, it leaves *V* as is, and (iii) when the execution of T_u and T_s cannot be serialized in any way, it interrupts the program. In cases (i) and (ii), the race has been tolerated; in case (iii) the race induces a sequentially inconsistent execution and, therefore, ToleRace is unable to handle it.

ToleRace has several shortcomings. First, a race type of case (iii) cannot be handled adequately: leaving version *V* or *V'* produces an inconsistent execution (a detailed example is described in [22]). Second, when the critical section contains multiple variables and accesses, the analysis of what race case it is can become complicated. Third, analysis of access patterns is either conservative (if

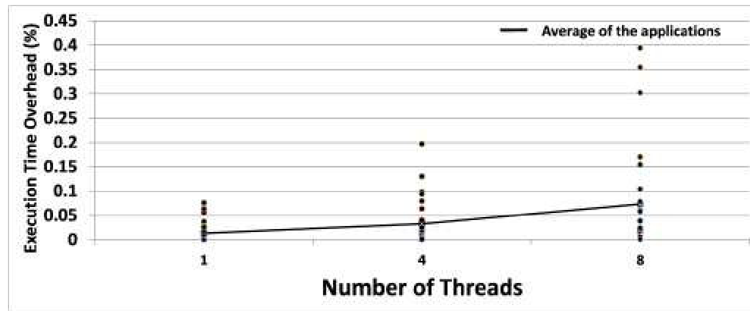


Figure 9. Execution time overhead of Pacman.

```
T1                                          T2

void ComputeSubTreeCosts(...) {
  ...                                        ...
 pb=b->parent;
  ...                      void ComputeSubTreeCosts(...)

Lock
 pb->subtree_cost+=b->subtree_cost;
 pb->interaction_synch +=1;                  ...
                            b->interaction_synch = 0;
                                             ...
Unlock                      b->subtree_cost += b->cost;
}                                            ...
```

Figure 10. Race discovered in FMM.

static) or slow (if dynamic). Finally, comparisons and copies are slow and race-prone.

ISOLATOR [22] takes a different approach. When a safe thread T_s enters a critical section, it makes a copy in software of the pages that contain the shared variables that will be accessed in the critical section (*Shadow Pages*). Then, it changes the protection bits of the original pages to make them inaccessible. T_s operates on the shadow pages. If an unsafe thread T_u accesses the original pages, it gets an exception and gets de-scheduled. When T_s leaves the critical section, it copies the shadow pages back to the original pages and unprotects the latter.

ISOLATOR has the advantage of always producing consistent executions. In addition, thanks to an optimization, the number of page copies can be reduced. However, it has several shortcomings. The first one is the substantial compiler and operating system (OS) support (or code re-writing by the user) required to place variables in the correct pages and adapt to changing access patterns in the program. To apply ISOLATOR to PARSEC, we would have to rewrite the code and change the variable allocation significantly.

A second shortcoming is that, if such rewriting is not provided, ISOLATOR will often need to copy large amounts of data at critical section entries and exits. For example, such data copying is the main reason why ISOLATOR reports up to 8x overhead for the microbenchmarks in [22]. Finally, ISOLATOR is prone to deadlocks and livelocks due to false sharing at page level — e.g., assume that unsafe thread T_u gets de-scheduled and then T_s attempts to access a variable in a page that T_u has protected. Moreover, the timeout-based mechanism that is used to detect such deadlocks is very slow.

Overall, neither ToleRace nor ISOLATOR provides the desired solution to handle asymmetric races.

7.1.1. Summary of Pacman's Advantages

Pacman addresses limitations of these schemes. First, Pacman has negligible execution overhead and can be used in production runs because it (1) does not perform any data copying (unlike ISOLATOR and ToleRace) and (2) minimizes the stall time of unsafe threads by accurately identifying the addresses where safe and unsafe threads conflict. This last property results from detecting conflicts dynamically in hardware, while the other schemes use conservative, static software analysis to predict conflicts (or slow, dynamic software analysis). Second, Pacman does not need any support from compiler or (in the baseline design without virtualization) OS, or source code modifications (although it may change the code inside synchronization macros). This is in contrast to the other schemes, which rely on the compiler (to identify shared accesses inside critical sections, perform code transformations, or associate variables with locks), OS (to store variables protected by a given lock in the same page), and source code modifications. Finally, Pacman is not prone to wasting time on deadlocks like ISOLATOR (due to the latter's page-level false sharing and slow timeout mechanism) and cannot create inconsistent executions like ToleRace.

Pacman can potentially break the atomicity of a critical section in some rare cases. These cases may occur when Pacman breaks a deadlock (Section 3.4.3) and when a thread migrates while it is executing a critical section (Section 5.2). The cases in the second group are Pacman's false negatives.

7.2. Other Related Work

Pacman is related to Transactional Memory (TM) [10] in that it presents a concept analogous to strong atomicity [3] between a transaction and a non-transactional access. However, Pacman operates on lock-based code. Moreover, compared to HTM, Pacman does not need speculation, rollback, timestamp support, or version management. Even to detect inter-thread conflicts, Pacman cannot leverage HTM's tagging of cache lines: since Pacman is non-speculative, data can overflow into memory. Hence, Pacman needs to keep a SigTable in memory. Compared to STM, Pacman does not need to analyze the code.

Pacman is also related to hardware-based mechanisms for fine-grain memory protection, such as UFO [1] and iWatcher [33]. In UFO, each memory line has some bits that specify protection information. Such bits travel with the line to caches. It is possible to support Pacman-like functionality with UFO. However, UFO is substantially more intrusive, as it requires maintaining these distributed bits and building exception handlers for them. iWatcher is similar although it targets uniprocessors.

Pacman is also related to the many software or hardware schemes that detect and avoid atomicity violations, such as AVIO [11], AtomAid [12], AtomTracker [15], or LifeTx [31]. While Pacman focuses on avoiding races rather than atomicity violations, its hardware is effectively being used to keep atomicity, albeit for only user-defined critical sections. As a result of the latter, Pacman needs no training runs. Finally, there are some software-only schemes to tolerate races and bugs, such as Rx [21] or Frost [28]. Such techniques, while effective, have substantially higher overheads. We find Pacman to have negligible overhead.

8. Conclusions

This paper proposed Pacman, the first scheme designed to tolerate asymmetric data races in production runs with negligible execution overhead. Pacman leverages cache coherence hardware to temporarily protect the variables that a thread accesses in a critical section. Unlike the previous, software-based schemes, Pacman induces negligible slowdown, needs no compiler or (in the baseline design) OS support, and requires no application source code

changes — although small changes are needed in some libraries. Moreover, its hardware is unintrusive since it is concentrated in a module in the network, rather than in the cores. We evaluated Pacman for SPLASH-2, PARSEC, Sphinx3, and Apache and showed that it has negligible overhead. Moreover, we uncovered two unreported asymmetric data races.

Pacman provides a hardware primitive for dynamically and selectively preventing accesses by certain processors to a set of addresses. This primitive can have several uses in performance and correctness debugging. We are now exploring such uses.

Acknowledgments

We thank the anonymous reviewers, Satish Narayanasamy, and the I-ACOMA group members for their comments.

References

[1] L. Baugh, N. Neelakantam, and C. Zilles. Using hardware memory protection to build a high-performance, strongly-atomic hybrid transactional memory. In *ISCA*, June 2008.

[2] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM*, 13(7), 1970.

[3] C. Blundell, E. Lewis, and M. Martin. Subtleties of transactional memory atomicity semantics. *Comp. Arch. Letters*, 5(2), November 2006.

[4] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas. Bulk disambiguation of speculative threads in multiprocessors. In *ISCA*, June 2006.

[5] D. E. Culler, A. Gupta, and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., 1997.

[6] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *Comput. Sci. Eng.*, 1998.

[7] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *SOSP*, October 2003.

[8] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In *OSDI. Slides in usenix.org/event/osdi10/tech/slides/erickson.pdf*, October 2010.

[9] M. A. Heinrich. *The performance and scalability of distributed shared-memory cache coherence protocols*. PhD thesis, Stanford University, CA, USA, 1999.

[10] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, May 1993.

[11] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting atomicity violations via access interleaving invariants. In *ASPLOS*, October 2006.

[12] B. Lucia, J. Devietti, K. Strauss, and L. Ceze. Atom-Aid: Detecting and surviving atomicity violations. In *ISCA*, June 2008.

[13] E. Lusk, J. Boyle, R. Butler, et al. *Portable programs for parallel processors*. Holt, Rinehart & Winston, 1988.

[14] S. L. Min and J.-D. Choi. An efficient cache-based access anomaly detection scheme. In *ASPLOS*, April 1991.

[15] A. Muzahid, N. Otsuki, and J. Torrellas. AtomTracker: A Comprehensive Approach to Atomic Region Inference and Violation Detection. In *MICRO*, December 2010.

[16] A. Muzahid, D. Suarez, S. Qi, and J. Torrellas. SigRace: Signature-based data race detection. In *ISCA*, June 2009.

[17] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *PLDI*, June 2006.

[18] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. In *PLDI*, June 2007.

[19] M. Prvulovic. CORD: Cost-effective (and nearly overhead-free) order-recording and data race detection. In *HPCA*, June 2006.

[20] M. Prvulovic and J. Torrellas. ReEnact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *ISCA*, June 2003.

[21] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies—a safe method to survive software failures. In *SOSP*, October 2005.

[22] S. Rajamani, G. Ramalingam, V. P. Ranganath, and K. Vaswani. ISOLATOR: Dynamically ensuring isolation in concurrent programs. In *ASPLOS*, March 2009.

[23] P. Ratanaworabhan, M. Burtscher, D. Kirovski, B. Zorn, R. Nagpal, and K. Pattabiraman. Detecting and tolerating asymmetric races. In *PPoPP*, February 2009.

[24] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC simulator, January 2005. http://sesc.sourceforge.net.

[25] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam. Implementing signatures for transactional memory. In *MICRO*, December 2007.

[26] R. Sasanka, M.-L. Li, S. V. Adve, Y.-K. Chen, and E. Debes. ALP: Efficient support for all levels of parallelism for complex media applications. *ACM TACO*, 4, March 2007.

[27] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, November 1997.

[28] K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy. Detecting and surviving data races using complementary schedules. In *SOSP*, October 2011.

[29] J. Wu, H. Cui, and J. Yang. Bypassing races in live applications with execution filters. In *OSDI*, October 2010.

[30] J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multiprocessor. In *ISCA*, June 2009.

[31] J. Yu and S. Narayanasamy. Tolerating Concurrency Bugs Using Transactions as Lifeguards. In *MICRO*, 2010.

[32] W. Zhang, C. Sun, and S. Lu. ConMem: Detecting severe concurrency bugs through an effect-oriented approach. In *ASPLOS*, March 2010.

[33] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iWatcher: Efficient architectural support for software debugging. In *ISCA*, June 2004.

[34] P. Zhou, R. Teodorescu, and Y. Zhou. HARD: Hardware-assisted lockset-based race detection. In *HPCA*, February 2007.